Corbin Mayes - 2/14/19

Sudoku Assignment

Initialization

To begin the sudoku assignment, I began by writing the SAT class that is called in solve_sudoku. The initializer takes in the cnf file that is passed in and begins by reading the lines of the file then passing those lines into a list. Each line is then broken up into another list so the first layer of the lists is each whole clause then the second layer is all the literals within that clause. Afterwards, it reads through the list and adds all the literals to a dictionary with numbered keys and then adds the keys to a list. This just makes it easier to keep track of all the variables. Next I did the same thing with all the clauses and added them to a dictionary and list.

GSAT

To begin the GSAT function, the probability, max number of flips and max number of tries are all passed into the function. Then a for loop starts the function so GSAT will run the number of tries or until a solution is found. For each time through GSAT, a random truth assignment is generated with each variable being given a random boolean value. Then another for loop is added to flip as many variables as is allowed by the max number of flips for each truth assignment attempt. It first tests if the truth assignment satisfies all the clauses and if it does then return true and create the solution list.

Satisfy function

> This function takes the truth assignment passed into it and tests its assignment of all the literals within the clause. If a clause is not satisfied by literals within the truth assignment then the function stops and returns False. Otherwise, if all clauses are satisfied, the function returns True.

After it tests for satisfaction, a variable is chosen based on the probability passed in to be either a random variable or the variable that when changed would minimize the most clause failures.

var_to_flip function

> This function iterates through every literal in the truth assignment and flips their current boolean assignment. Then it passes the new truth assignment into the function num_clause_fails. If the the number of clause fails is less than the current min then the new literal becomes the min. If the number of clause fails is equal to the current min then it appends it to the list of mins. The literal is then switched back to its original state. Then it returns a random min from the list.

num_clause_fails function

> This function takes the truth assignment given it and counts how many clause failures the current assignment has. To write this, I took the code from the satisfy function and removed the early cutoff for a counter instead.

After a variable has been picked, the variable is then flipped within the current truth assignment.

WalkSAT

For WalkSAT I copied over GSAT to begin with because it is in essence the same with a few slight differences. First it does not have a for loop at the beginning because it only does one attempt instead of multiple. Next WalkSAT picks a variable from a single unsatisfied clause instead of the entire list of variables.

rand_false_clause_find function

> This iterates through all the clauses and adds to the list all the unsatisfied ones for the current truth assignment. Then it returns a random clause from the list

clause_var_to_flip function

> Almost exactly the same as var_to_flip except it only searches for a variable in the given clause.

Corbin Mayes - 2/14/19

MapColor

To make the WalkSAT able to solve the map coloring problem, I manually created a cnf file that contains all the clauses required for the problem. Then I created a solve_MapColor test function that runs similarly to solve_sudoku because I wanted everything to be in a separate place so not to possible interfere

---

DPLL

Initialization

To initialize the DPLL class, I copied over some of the code from the SAT class initializer. Similarily it takes the cnf and converts it into a set of frozensets where each frozenset is a clause and the contents in each frozenset are the literals.

DPLL Recursion

The goal of this function is to reduce the set of clauses to an empty set by simplifying the set each time the function recurses. If an empty set is reached then the set of clauses is therefore satisfiable. The function starts by testing if the set is already empty and therefore satisfiable. If it isn't immediately then it checks for unit clauses.

is_unit_clause function

> This iterates through the set of clauses passed into the function and if it finds a unit clause, a clause with only one literal, the it returns the unit clause

If a unit clause is found within the set then it tests first if the unit clause of the negated literal is also within the set of clauses because if so then the set of clauses is unsatisfiable and the function ends. If it is not, then the simplify function is called and it passes in both the literal and the set

simplify function

> This function simply goes through the set of clauses and removes any clause containing the literal passed into the function from the set because that clause has been satisfied. It also removes the negated literal from any clause that contains the negated literal because that literal can not be satisfied in that clause assuming the literal is true. Then it returns the new altered set of clauses

After the first round of unit clauses has been removed from the set, it tests if the set is satisfiable again. If not then it picks a random literal from the shortest clause in the set because we want to try to quickly reduce that clause to a unit clause.

rand_lit_from_short_clause function

> This iterates through all the clauses in the set to find the shortest one. From the shortest clause it returns a random variable

It then simplifies the set of the literal picked and recursively calls dpll with the new simplified set. If the recursion is satisfiable, then the function returns satisfiable otherwise it recursively calls dpll with the simplified set of the negated variable and returns the result of that call.